TÌM KIẾM CHUỖI

Challenge 01

Phan Lộc Sơn
19120033

Đoàn Kim Huy
19120239

Đỗ Hoài Nam
19120296

Trần Anh Huy
19120082

Cấu trúc Dữ liệu và Giải thuật
19CNTN

Khoa Công nghệ Thông tin
Đại học Khoa học Tự nhiên
Việt Nam
23 - 11 - 2020

Mục lục

Phần I
TÌM KIẾM CHUỖI

1 Xác định vấn đề

Tìm kiếm chuỗi hiện diện rất nhiều trong cuộc sống, ví dụ như: tìm tên thầy dạy DSA trong danh sách giảng viên, tìm tên trong bảng điểm,.. hoặc trong khoa học, như là tìm liệu cấu trúc ADN của virus này có trong virus khác hay không.

Trong Tin học, các trình soạn thảo văn bản thường phải tìm kiếm (tất cả) các lần xuất hiện của một đoạn văn bản trong một văn bản dài. Thông thường văn bản được chỉnh sửa liên tục, và các phần văn bản cần tìm kiếm thì được nhập bởi người dùng. Việc phát minh ra các thuật toán tìm kiếm chuỗi hiệu quả đã giúp ích rất nhiều cho các bài toán kể trên.

Bài toán tìm kiếm chuỗi thường được mô tả theo mô hình sau:
Một chuỗi cần tìm kiếm S có dạng một chuỗi kí tự S[1..m], cần tìm chuỗi S trong một đoạn văn bản T[1..n], với m <= n. Đồng thời, tất cả các ký tự trong T và S đều thuộc về một tập hữu hạn các ký tự cho trước. Chuỗi S được gọi là xuất hiện bắt đầu từ vị trí p + 1 nếu thoả điều kiện: T[p + j] = S[j], với 1 <= j <= m.

Có 3 thuật toán thường được dùng để tìm kiếm chuỗi:

- Thuật toán Brute-force (vét cạn, hay còn gọi là thuật trâu)
- Rabin - Karp
- Knuth - Morris - Pratt

Chi tiết của từng thuật toán sẽ được trình bày bên dưới.

2 Một số thuật toán tìm kiếm chuỗi

2.1 Brute-force

Giải thuật Brute-Force, hay còn gọi là vét cạn, là thuật toán đơn giản nhất trong các thuật toán tìm kiếm chuỗi con *pattern* trong chuỗi cha *text*.

Có thể giải thích đơn giản, giải thuật Brute-Force so sánh lần lượt mỗi chuỗi con *subtext* của *text* có cùng chiều dài với *pattern* với *pattern*, nếu tìm được, trả về kết quả là vị trí được tìm thấy; khi không tìm được kết quả mong muốn, trả về giá trị quy ước là không tìm thấy.

Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
text = `Let them go!`
pattern = `them`

Let them go!
them

Let them go!
 them

Let them go!
  them

Let them go!
  them

Let them go!
   them

Let them go!
    them

Let them go!
    them

Let them go!
    them

Let them go!
    them

Ta tìm thấy chuỗi pattern tại vị trí thứ 4!

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Brute-Force:

```
function: Brute-Force String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  vị trí tìm thấy = -1
  subtext = chuỗi con đầu text có độ dài bằng pattern
  while (chưa tìm thấy hoặc chưa tới cuối text)
    if (từng ký tự của subtext = pattern):
      trả về vị trí
    else:
      dịch chuyển chuỗi con subtext trong text sang phải 1 chữ cái
  Trả về: vị trí tìm thấy
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp N - M + 2 chuỗi *subtext* đầu không trùng với *pattern*)

  → Cần M(N - M + 1) lần. Vì duyệt tới cuối mảng nên đây là trường hợp tìm thấy ở cuối mảng, hoặc không tìm thấy
  → Cận trên $O(MN)$ (vì N > N - M + 1).
  → Cấp phát bộ nhớ: 0.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự.

→ Cần M lần.
→ Cận trên $O(M)$.
→ Cấp phát bộ nhớ: 0.

Độ phức tạp của thuật toán trong trường hợp trung bình: $O(MN)$
Đánh giá:

- Dễ hiểu, thuật toán này chỉ duyệt từ đầu đến cuối, so sánh tuần tự từng chuỗi con với chuỗi cần tìm kiếm.
- Không cần bước tiền xử lý (như các thuật toán được trình bày bên dưới).
- Độ phức tạp $O(MN)$. Không cần xin thêm bộ nhớ.

2.2 Rabin-Karp

Thuật toán Rabin-Karp là một thuật toán được sử dụng để tìm kiếm chuỗi con *pattern* trong chuỗi cha *text* bằng cách sử dụng một hàm băm.

Hàm băm là một hàm chuyển đổi mọi chuỗi thành giá trị số, giá trị này được gọi là mã băm của nó. Ví dụ, chúng ta có thể có hàm băm hash("hello")=5.

Giống như Thuật toán Brute-Force, thuật toán Rabin-Karp cũng dịch *pattern* qua từng phần tử trong *text* để so sánh. Nhưng sự khác biệt là thuật toán Rabin-Karp so khớp mã băm của *pattern* với mã băm của chuỗi con *subtext* của *text*, và nếu mã băm khớp thì thuật toán sẽ so sánh từng ký tự trong 2 chuỗi với nhau.

Nếu mã băm được biểu diễn bằng số nguyên không quá 64 bits, độ phức tạp thời gian (time-complexity) của việc so sánh *pattern* có độ dài m với *subtext* có cùng độ dài giảm từ O(m) xuống O(1).

Tuy nhiên mọi thứ đều có hai mặt, vấn đề của hàm băm đó là mã băm của hai chuỗi khác nhau có thể bằng nhau. Ví dụ xét hàm băm hash(S) tính mã băm của xâu S bằng cách cộng mã ASCII của các kí tự trong S: hash("abcd")=97+98+99+100=394, hàm băm này quá đơn giản và có khả năng gây trùng mã băm cao: hash("dcba")=100+99+98+97=394, nhưng "abcd" $\neq$ "dcba".

Một hàm băm tốt thoả mãn các điều kiện sau:

- Tính toán nhanh.
- Xác suất trùng mã băm nhỏ.

Thuật toán Rabin-Karp xây dựng hàm băm với ý tưởng cơ số: xem mọi xâu như là một chuỗi số với một cơ số (base) nào đó. Hàm băm được tính tương tự như việc ta chuyển một số nguyên về giá trị của nó, nếu là xâu kí tự thì có thể sử dụng mã ASCII (hoặc UNICODE). Một số ví dụ:

- base=10, hash("425")=4×102+2×101+5×100=425.
- base=26, kí tự là chữ cái từ a đến z: hash("abc")=97×262+98×261+99×260=68219.

Để tránh tràn số thì kết quả trên được chia lấy dư cho một số q, thường được chọn là một số nguyên tố lớn. Nếu gọi tập các kí tự được sử dụng trong chuỗi là $\sum$ thì base thường được chọn sao cho base=$|\sum|$ hoặc là một số nguyên tố lớn.

Độ phức tạp thời gian để tính mã băm của chuỗi độ dài k mất O(k). Khi hiện thực thuật toán, ta sẽ "trượt" *pattern* có độ dài m trên *text* từ vị trí 0 đến n-m để so sánh mã băm. Rolling hash là hàm băm có thể tính mã băm h_i của *text*[i... i+m-1] dựa trên mã băm h_{i-1} của *text*[(i-1)... (i+m)] chỉ trong thời gian O(1) thay vì tính lại trong thời gian O(m), từ đó tăng tính hiệu quả.

$h_i = (base \times (h_{i-1} - base^{m-1} \times text[i-1]) + text[i+m-1])\%q$

Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
Để đơn giản, ta chọn tập các kí tự được sử dụng trong chuỗi là {a, b, c, d, e, f, g, h, i, j} ứng với giá trị số lần lượt là {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} và base=10, q=13.
text = abccddaefg
pattern = cdd
Mã băm của *pattern*: hash("cdd")=$(3 \times 10^2 + 4 \times 10^1 + 4 \times 10^0)\%13 = 344\%13 = 6$

abccddaefg // hash("abc")=$(1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0)\%13 = 123\%13 = 6$
cdd // Vì mã băm của *subtext* "abc" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"≠*pattern* nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("bcc")=$(10 \times (123 - 10^2 \times 1) + 3)\%13 = 233\%13 = 12$
cdd // Vì mã băm của *subtext* "bcc" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("ccd")=$(10 \times (233 - 10^2 \times 2) + 4)\%13 = 334\%13 = 9$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("cdd")=$(10 \times (334 - 10^2 \times 3) + 4)\%13 = 344\%13 = 6$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"=*pattern* nên chuỗi đã được tìm thấy.

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Rabin-Karp:

```
function: Rabin-Karp String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  n=chiều dài chuỗi text
  m=chiều dài chuỗi pattern
  Tính mã băm của pattern và chuỗi con đầu text có độ dài bằng pattern
  Dịch pattern từ vị trí 0, qua từng phần tử của text đến vị trí n-m
    Kiểm tra mã băm của subtext hiện tại và pattern
      Nếu bằng nhau, so sánh từng ký tự trong 2 chuỗi
        Nếu hoàn toàn giống nhau, trả về vị trí
    Tính mã băm của subtext tiếp theo, nếu âm cộng thêm một lượng q
  Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả về -1
     báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N:
Phân tích độ phức tạp của thuật toán:

Trong quá trình tiền xử lý, để tính mã băm cho *pattern* cần duyệt qua M phần tử, tương tự với *subtext* đầu tiên *text*, vậy độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*, khi đó chỉ cần tốn M lần so sánh các ký tự, kết hợp với quá trình tiền xử lý nên độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp mã băm bằng nhau và cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp chuỗi cần tìm nằm cuối, N - M + 2 chuỗi *subext* đầu không trùng với *pattern*).

  → Cần nhiều nhất M(N - M + 1) lần so sánh. Vì duyệt tới cuối chuỗi nên đây là trường hợp tìm thấy ở cuối hoặc không tìm thấy và xảy ra trùng mã băm ở tất cả lần so sánh trước đó.
  → Cận trên $O(MN)$ (vì N > N - M + 1).

  Trường hợp trung bình: $O(N + M) \in O(M)$

Do trong quá trình tìm kiếm có thực hiện tính toán và lưu mã băm nên chi phí bộ nhớ là hằng số $O(1)$. Đánh giá:

- Mặc dù trong trường hợp xấu nhất độ phức tạp là $O(MN)$ không tốt hơn giải thuật Brute-Force, nhưng giải thuật Rabin-Karp hoạt động tốt hơn nhiều trong trường hợp trung bình và thực tế.

- Có bước tiền xử lý với độ phức tạp $O(M)$ trước khi bắt đầu tìm kiếm.
- Độ phức tạp $O(MN)$. Chi phí bộ nhớ $O(1)$.

2.3 Knuth-Morris-Pratt

Giải thuật Knuth-Morris-Pratt, về cơ bản cũng giống như thuật toán Brute-Force, tuy nhiên, chỉ khác ở chỗ, Brute-Forch khi so sánh *pattern* và *subtext*, Brute-Force so sánh toàn bộ các ký tự của chúng lại từ đầu, còn Knuth-Morris-Pratt, từ lần so sánh trước, sẽ quyết định có so sánh pattern với *subtext* kế tiếp không, và nếu không sẽ shift bao nhiêu bước để đến lượt so sánh tiếp theo.

Để dễ hiểu, ta xét 1 ví dụ như sau:
text = ababcababd
pattern = ababd
Quy ước vị trí ký tự đầu tiên trong chuỗi là 1.
Đầu tiên, tính giá trị của mảng Failure Function:
Failure Function tại vị trí i là độ dài của chuỗi thoả mãn:

- Chuỗi phải là dài nhất có thể.
- Là chuỗi prefix của text[0, i].
- Là chuỗi suffix của text[1, i].

Như vậy ta tính được: f = [0, 0, 1, 2, 0]

---

ababcababd
ababd // Bắt đầu so sánh tương tự giải thuật Brute-Force.

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd // Không khớp nhau, shift sang phải để tìm kiếm
// So sánh thất bại ở vị trí thứ 5, ta chú ý tới f[4] = 2 > 0
// KMP nói rằng để tối ưu hoá thời gian, ta cần shift tiếp 2 bước.
// Để ý rằng f[2] = 0, nên ký tự 1 khác ký tự 2
// Mà pattern[1] = text[1], pattern[2] = text[2], pattern[1] != pattern[2]
// => pattern[1] != text[2] => Việc shift 1 bước rồi so sánh là dư thừa.

---

ababcababd
ababd // Vì ta đã biết f[4] = 2 -> pattern[1, 2] = pattern[3, 4]
// Ở lượt trước, ta có pattern[1, 2] = text[1, 2] và pattern[3, 4] = text[3, 4]
// Ngoài ra, vừa rồi ta shift 2 bước -> pattern[1, 2] = text[3, 4]
// Tức là ta không cần so sánh 2 ký tự đầu ở lượt này
// Vị trí không khớp là *pattern*[3], cùng với f[2] = 0 nên
// Bước tiếp theo cần shift sang phải 1 ký tự để tiếp tục so sánh.

---

ababcababd
ababd // Tiếp tục so sánh *pattern* với *text* tương tự giải thuật Brute-Force.

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd

```
    ababd
ababcababd
    ababd       // pattern khớp subtext tại vị trí thứ 6 của text
              // Vị trí tìm được là 6, kết thúc tìm kiếm.
```

Từ ví dụ trên, ta tìm cách thiết kế ý tưởng của KMP để tìm chuỗi 1 cách tổng quát:

- Ta thấy rằng, để KMP hoạt động, cần phải xử lý trước pattern, chính là tìm Failure Function
- Trong khi tìm kiếm chuỗi, ta có thể dùng 2 biến i và j dùng để duyệt qua text và pattern để so sánh các ký tự trong chúng:
  1. Nếu pattern[j] = text[i], tăng thêm i và j để so sánh ký tự kế tiếp.
  2. Nếu pattern[j] $\neq$ text[i], và j = 1, tức là ký tự đầu của chúng khác nhau, ta chỉ cần shift chuỗi so sánh chuỗi con kế tiếp.
  3. Nếu pattern[j] $\neq$ text[i], và j $>$ 1, tức là đã có 1 lượng j - 1 ký tự ở trước đã được so sánh đúng, ta lấy f[i - 1] và và shift chuỗi với số bước là min((i - 1) - f[i - 1], 1) để chuẩn bị cho lượt so sánh kế tiếp.

Từ ý tưởng trên, ta xây dựng mã giả cho thuật toán KMP:

```
function: Build Failure Function
pattern: chuỗi cần tìm
Trả về: Mảng f lưu độ dài lớn nhất f[i] của chuỗi chung prefix và suffix của
   pattern[1, i].

  f[] = 0 để lưu mảng trả về
  fpre = 0;   // lưu f[i - 1] trong vòng lặp bên dưới
  i = 2;    // Ta chỉ tính f[2] trở đi, f[1] luôn bằng 0
  while (i <= n)
    Nếu pattern[i] == pattern[f[i - 1] + 1]
      fpre++
      f[i] = fpre // tận dụng kết i - 1 kết quả so sánh của f[i - 1], ta chỉ
         cần so sánh 1 ký tự.
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] = 0
      // Tức là suffix và prefix là chuỗi có độ dài f[i - 1] + 1 có ký tự
         cuối không giống
      // -> f[i] = 0
      f[i] = 0
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] > 0
      // Trường hợp này, có thể f[i] = f[i - 1]
      // Ta sẽ tận dụng pattern[1, f[i- 1]] = pattern[i - f[i-1], i]
      // Và f[f[j]] (độ dài này lớn nhất là j - 1)
      // Ví dụ AAABAAAA khi xét i = 8, ta lợi dụng chuỗi [AA]ABAAAA =
         A[AA]BAAAA và [AAA]BAAAA = AAAB[AAA]A
      // Suy ra được [AA]ABAAAA = A[AA]BAAAA = AAABA[AA]A
      // Như thế chỉ cần xét ký tự AA[A]BAAAA và AAABAAA[A] để xác định f[i]
      fpre = f[fpre]
  Trả về: f[]

function: KMP String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  i = 1; j = 1;
  while (i < len(text) - len(pattern))
    Nếu pattern[j] == text[i]
      Tiếp tục i++ và j++ và so sánh
        Nếu j = len(pattern) trả về đã tìm thấy pattern tại vị trí i - j.
    Nếu pattern[j] != text[i] và j == 1
      i = i + 1  // shift sang phải 1 bước
    Nếu pattern[j] != text[i] và j > 1
      j = f[i] và tiếp tục so sánh text[i] và pattern[j]
```

```
41          Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả về -1
               báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Trong quá trình tiền xử lý (tính hàm Failure Function), trong vòng while chỉ tốn nhiều nhất 1 - 2 lần để tính f[i], vậy độ phức tạp là $O(M)$
- Trong quá trình so sánh, mỗi lần, ta hoặc sẽ tăng i lên 1 đơn vị, hoặc sẽ giảm j ít nhất 1 đơn vị, nghĩa là tăng k lên 1 đơn vị, với k = i - j < N. Vậy tổng cộng trong vòng lặp có thể lặp < 2n lần (i và k không thể vượt quá N), như vậy độ phức tạp là $O(2N)$.
  → Độ phức tạp $O(M + N)$.
  → Cấp phát bộ nhớ: $O(M)$.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự. Ngoài ra còn tốn thêm chi phí tính Failure Function $O(M)$.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: $O(M)$.

Độ phức tạp của thuật toán trong trường hợp trung bình:

- Cận trên $O(N + M)$.
- Cấp phát bộ nhớ: $O(M)$.

Đánh giá:

- Nhanh, chi phí tuyến tính.
- Phải xử lý trước khi tìm kiếm thực sự.
- Độ phức tạp $O(M + N)$. Chi phí bộ nhớ $O(M)$.

3 So sánh các thuật toán tìm kiếm chuỗi

Thuật toán	Tiền xử lý	Tìm kiếm
Brute-Force	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Ngoại trừ thuật toán Brute-Force, hai thuật toán còn lại đều có bước tiền xử lý trước khi bắt đầu tìm kiếm. Thời gian chạy của từng thuật toán là tổng của thời gian tiền xử lý và thời gian tìm kiếm.

- Do không có bước tiền xử lý, thuật toán Brute-Force chạy khá chậm khi phải so sánh lần lượt từng vị trí trong chuỗi.
- Mặc dù trong trường hợp xấu nhất, thuật toán Rabin-Karp cũng không nhanh hơn Brute-Force nhưng trong thực tế lại hoạt động hiệu quả với độ phức tạp tuyến tính.
- Knuth-Morris-Pratt là thuật toán tốt nhất trong 3 thuật toán nêu trên vì có độ phức tạp tuyến tính trong tất cả trường hợp (khắc phục được điểm yếu của Rabin-Karp khi gặp trường hợp xấu nhất).

Phần II
TÌM KIẾM CHUỖI TRONG TIN HỌC

Ví dụ về Crossword game

1 Hướng giải Crossword game

Trong phần này, nhóm chúng tôi quyết định sẽ sử dụng thuật toán KMP để làm giải thuật tìm kiếm chuỗi, bởi vì giải thuật này có vẻ nhanh hơn 2 giải thuật còn lại với thời gian $O(M + N)$.

Thuật toán chính của chương trình là đọc mảng chữ cái vào mảng **table** để tìm theo hàng ngang, và tạo mảng **Transpose_table** bằng cách chuyển vị mảng **table** để tìm theo hàng dọc.

Tìm bằng thuật toán KMP, hàng ngang trước và hàng dọc sau, nếu tìm thấy, in ra vị trí tìm thấy, nếu không, in ra NF.

Độ phức tạp:

- Có H hàng, mỗi hàng cần tìm 1 chuỗi có độ dài M trong chuỗi W, vậy độ phức tạp là $O(H * (M + W))$.
- Có W cột, mỗi cột cần tìm 1 chuỗi có độ dài M trong chuỗi H, vậy độ phức tạp là $O(W * (M + H))$.
  $\rightarrow$ Như vậy, độ phức tạp là: $O(2WH + M(H + W))$ đối với từng từ cần tìm kiếm.

2 Một số mẫu kiểm tra lời giải

Dưới đây là bảng số liệu nhóm chúng tôi đã thực hiện trên các tập dữ liệu $n \times n$: 5, 10, 20, 50:

- Bộ dữ liệu ngẫu nhiên:

Size	BF	RK	KMP
5	125	178	146
10	135	147	168
20	264	175	198
50	649	218	181
100	1533	326	203
600	15767	662	287

Bảng 1: Số liệu với bộ test được sinh ngẫu nhiên, đơn vị micro second.

Nhận xét: Ở các bộ dữ liệu nhỏ, Naive nhanh hơn Rabin-Karp và Knuth-Morris-Pratt vì không có bước tiền xử lý.

- Bộ dữ liệu xấu nhất:

Size	Naive	Rabin-Karp	Knuth-Morris-Pratt
5	747	93	129
10	1052	102	148
20	3246	125	139
50	8796	222	205
100	20579	579	253
600	148412	1247	326

Bảng 2: Bảng số liệu với bộ test được sinh xấu nhất, đơn vị micro second.

Nhận xét: Tại bộ dữ liệu xấu nhất, Knuth-Morris-Pratt và Rabin-Karp nhanh hơn Naive
Ngoài ra, càng tăng size bộ dữ liệu, tốc độ tăng thời gian của Naive > Rabin-Karp > Knuth-Morris-Pratt (Naive tăng nhanh nhất, Knuth-Morris-Pratt tăng chậm nhất)

Tài liệu

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 2009 *Introduction to Algorithms (3rdedition).* Massachusetts Institute of Technology

[2] Rabin-Karp Algorithm,
texttthttps://www.programiz.com/dsa/rabin-karp-algorithm

[3] Rabin-Karp Algorithm for Pattern Searching,
texttthttps://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching